Fundamentals of Computer Science I: Media Computing (CS151.02 2007F)

Laboratory: Writing More Complex Color Transformations

Summary: In the past few laboratories, you've learned how to write filters using basic color transformations and how to write your own color transformations. In the most recent reading, you've learned about Scheme's primary numeric procedures. In this lab, you will further explore these procedures as you write some more complex color transformations.

Contents:

Preparation

a. Load a moderate-sized image (no more than about 250x250) and name the loaded image `source`.

b. Create and show a new 3x3 image and name it `canvas`.

c. Zoom in on `canvas`.

d. Fill in the pixels of `canvas` as follows. Note that you may want to open this lab in a Web browser and cut and paste.

```
(image.set-pixel! canvas 0 0 (rgb.new 192 0 0))
(image.set-pixel! canvas 1 0 (rgb.new 192 0 192))
(image.set-pixel! canvas 2 0 (rgb.new 0 0 192))
(image.set-pixel! canvas 0 1 (rgb.new 192 192 0))
(image.set-pixel! canvas 1 1 (rgb.new 0 192 192))
(image.set-pixel! canvas 2 1 (rgb.new 255 255 255))
(image.set-pixel! canvas 0 2 (rgb.new 0 192 0))
(image.set-pixel! canvas 1 2 (rgb.new 0 0 0))
(image.set-pixel! canvas 2 2 (rgb.new 192 192 192))
```

e. Create and show a new 17x1 image and name it `blues`.

f. Fill in the pixels of `blues` as follows.

```
(image.set-pixel! blues 0 0 (rgb.new 0 0 0))
(image.set-pixel! blues 1 0 (rgb.new 0 0 16))
(image.set-pixel! blues 2 0 (rgb.new 0 0 32))
(image.set-pixel! blues 3 0 (rgb.new 0 0 48))
(image.set-pixel! blues 4 0 (rgb.new 0 0 64))
(image.set-pixel! blues 5 0 (rgb.new 0 0 80))
(image.set-pixel! blues 6 0 (rgb.new 0 0 96))
(image.set-pixel! blues 7 0 (rgb.new 0 0 112))
(image.set-pixel! blues 8 0 (rgb.new 0 0 128))
(image.set-pixel! blues 9 0 (rgb.new 0 0 144))
(image.set-pixel! blues 10 0 (rgb.new 0 0 160))
(image.set-pixel! blues 11 0 (rgb.new 0 0 176))
(image.set-pixel! blues 12 0 (rgb.new 0 0 192))
(image.set-pixel! blues 13 0 (rgb.new 0 0 208))
(image.set-pixel! blues 14 0 (rgb.new 0 0 224))
(image.set-pixel! blues 15 0 (rgb.new 0 0 240))
(image.set-pixel! blues 16 0 (rgb.new 0 0 255))
```

Exercises

Exercise 1: Using Modulo

As the reading suggests, the `modulo` procedure computes a value much like the remainder, except that the result is always the same sign as the second parameter, called the modulus. (So, when we use a positive modulus, we get a positive result.) The reading also suggests that `modulo` provides an interesting alternative to using `max` and `min` to limit the values of functions.

a. What value do you expect each of the following to produces?

```
> (modulo 254 256)
> (modulo 256 256)
> (modulo 257 256)
> (modulo 515 256)
> (modulo 2567 256)
> (modulo 0 256)
> (modulo -256 256)
> (modulo -257 256)
> (modulo -255 256)
> (modulo -1 256)
```

b. Check your answers experimentally, one at a time. If you find that any answers are incorrect, try to figure out why (asking me or a tutor if necessary), and then rethink your remaining answers before checking them experimentally.

c. Consider the following two named color transformations

```
(define my-double-1
  (lambda (color)
    (rgb.new (min (* 2 (rgb.red color)) 255)
             (min (* 2 (rgb.green color)) 255)
             (min (* 2 (rgb.blue color)) 255))))
(define my-double-2
  (lambda (color)
    (rgb.new (modulo (* 2 (rgb.red color)) 256)
             (modulo (* 2 (rgb.green color)) 256)
             (modulo (* 2 (rgb.blue color)) 256))))
```

Are there any pixels in `canvas` for which you expect the two transformations to behave differently? If so, what difference do you expect?

d. Check your answer experimentally with

```
> (image.show (image.map my-double-1 canvas))
> (image.show (image.map my-double-2 canvas))
```

Exercise 2: Modulo, Revisited

As you may recall, the `rgb.phaseshift` procedure takes each component and does one of two things. If the component is 128 or higher, it subtracts 128 from the component. If the component is less than 128, it adds 128.

It is possible to implement `rgb.phaseshift` without using a conditional. (In fact, it is implemented without a conditional.) How? Through a clever application of `modulo`.

a. Write your own version of `rgb.phaseshift`, called `my-phaseshift`.

b. Test `my-phaseshift` by applying it to each pixel in `blues`.

c. Test `my-phaseshift` by applying it to each pixel in `canvas`.

Exercise 3: Trigonometric Transformations

Don't worry if you don't know the meanings of the trigonometric functions for this exercise; the important point is to get you to think about how you deal with different ranges of numbers.

As you may have noted, we can get some "interesting" effects by applying traditional functions in untraditional ways. Let's see if the trigonometric functions `sin` and `cos` produce any interesting effects.

Traditionally, we prefer that each of these functions takes as input a number between 0 and 2*pi. (Yes, they work fine when the input is larger or smaller than that range, but we'll still find it more convenient to use that range.) Both functions return values between -1 and 1, inclusive.

If we are to use `sin` and `cosine` to transform color components, we'll need to find a way to convert the components, in the range 0 to 255 (inclusive) to a value in the range 0 to 2*pi, and the results back to a number in the range 0 to 255.

a. Write and test a procedure, `component-sin`, that takes as input a number between 0 and 255 and, using `sin`, computes a new number between 0 and 255. (You'll need to convert the input to a value in the range 0 to 255 and the result to a value in that range.)

b. Build a new version of `blues` using the following instruction:

```
(define newblues (image.map (lambda (color) (rgb.new (component-sin (rgb.red color)) (component-sin (rgb.green color)) (component-sin (rgb.blue color)))) blues))
```

c. What do you expect the result to look like?

d. Check your answer experimentally.

e. What do you expect to happen if you do something similar with `source`?

f. Check your answer experimentally.

Exercise 4: From Reals to Integers

This exercise does not require the construction or use of images or colors. Instead, it asks you to explore a few of the standard Scheme numeric functions.

As the reading on numbers suggests, Scheme provides four functions that convert real numbers to nearby integers: `floor`, `ceiling`, `round`, and `truncate`. The reading also claims that there are differences between all four.

To the best of your ability, figure out what each does, and what distinguishes it from the other three. In your tests, you should try both positive and negative numbers, numbers close to whole numbers and numbers far from whole numbers. (Numbers whose fractional part is 0.5 are about as far from a whole number as any real number can be.)

Once you have figured out answers, check, the notes on this problem.

Exercise 5: An Exponential Transformation

Consider the following somewhat similar functions:

```
(define f
  (lambda (component)
    (round (* 255 (expt (/ component 255) 3)))))
(define g
  (lambda (component)
    (round (/ (expt component 3) (expt 255 3)))))
```

a. Explain in your own words what these procedures seem to do to a color component (that is, a number between 0 and 255).

b. Suppose we transform `blues` using these two functions, as follows:

```
(define flues (image.map (lambda (color) (rgb.new (f (rgb.red color)) (f (rgb.green color)) (f (rgb.blue color))) blues)))
(define glues (image.map (lambda (color) (rgb.new (g (rgb.red color)) (g (rgb.green color)) (g (rgb.blue color))) blues)))
```

How do you expect `flues` and `glues` to differ?

c. Check your answer experimentally.

For Those with Extra Time

If you have extra time left at the end of this lab, you might try the exploration below or you might try one of these problems.

Extra 1: Rounding, Revisited

You may recall that we have a number of mechanisms for rounding real numbers to integers. But what if we want to round not to an integer, but to only two digits after the decimal point? Scheme does not include a built-in operations for doing so. Nonetheless, it is fairly straightforward.

Suppose we have a real value stored in *r*. Figure out how to change *r* so that it rounds to the nearest hundredth. For example,

```
> (define r 22.71256)
> r
22.71256
___ ; fill in your instructions
> r
22.71
> (define r 10.7561)
> r
10.7561
___ ; fill in your instructions
> r
10.76
```

You may also want to express your instructions as a Scheme function.

Explorations

Exploration 1: Modified Greyscale

In one of the exercises, you saw that it is possible to use `expt` to compress one end of the spectrum of colors and spread out another end of the spectrum. In a previous laboratory, you learned how to convert an image to greyscale.

So, let's put those two things together: First we'll convert to greyscale and then we'll use one of the `expt-based` transformations.

What purpose might the combination serve?

Find an image for which the combination provides particularly interesting results.

Notes

Notes on Exercise 4: From Reals to Integers

Here are how we tend to think of the four functions:

`(floor r)` finds the largest integer less than or equal to *r*. Some would phrase this as "`floor` rounds down".

`(ceiling r)` finds the smallest integer greater than or equal to *r*. Some would phrase this as "`ceiling` rounds up".

`(truncate r)` removes the fractional portion of *r*, the portion after the decimal point.

`(round r)` rounds *r* to the nearest integer. It rounds up if the decimal portion is greater than 0.5 and it rounds down if the decimal portion is less than 0.5. If the decimal portion equals 0.5, it rounds toward the even number.

```
> (round 1.5)
2
> (round 2.5)
2
> (round 7.5)
8
> (round 8.5)
8
> (round -1.5)
-2
> (round -2.5)
-2
```

It's pretty clear that `floor` and `ceiling` differ - If *r* has a fractional component, then `(floor r)` is one less than `(ceiling r)`.

It's also pretty clear that `round` differs from all of them, since it can round in two different directions.

We can also tell that `truncate` is different from `ceiling`, at least for positive numbers, because `ceiling` always rounds up, and removing the fractional portion of a positive number causes us to round down.

So, how do `truncate` and `floor` differ? As the previous paragraph implies, they differ for negative numbers. When you remove the fractional component of a negative number, you effectively round up. (After all, -2 is bigger than -2.3.) However, `floor` always rounds down.

Why does Scheme include so many ways to convert reals to integers? Because experience suggests that if you leave any of them out, some programmer will need that precise conversion.

 This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.